课程回顾

- 动态规划原理：
  - 备忘
- 动态规划问题：最长公共子序列、最优二叉搜索树

动态规划问题

- 钢条切割
- 矩阵链乘法的最优括号化
- 多边形的最佳三角剖分
- 最长公共子序列
- 最优二叉搜索树
- 0-1背包

0-1背包

- 0-1背包问题（0-1 knapsack problem，教材p243）：给定n个物品和一个容量为W的背包，第i个物品价值为v_i、重量为w_i，应当如何选择装入背包的物品使得总价值最大？（参数均为正整数）
- 0-1整数规划问题

穷举法共2^n种装包方式

$$
\begin{aligned}
\max \ & \sum_{k=1}^{n} v_k x_k \\
\text{s.t.} \ & \sum_{k=1}^{n} w_k x_k \leq W, \\
& x_k \in \{0,1\}, \quad k=1,...,n.
\end{aligned}
$$

0-1背包 (续)

1. 0-1背包子问题（最优子结构）

$$\max \sum_{k=1}^{\boxed{i}} v_k x_k$$

$$\text{s.t.} \quad \sum_{k=1}^{\boxed{i}} w_k x_k \leq \boxed{j},$$

$$x_k \in \{0,1\}, \quad k = 1, \ldots, \boxed{i}.$$

- 子问题：求解背包容量为j，可选物品编号为1, 2, ..., i时的0-1背包问题

  考虑是否放入编号为i的物品：

  $m[i, j] = \max(\underbrace{m[i\text{-}1, j]}_{\text{不装入}},\quad \underbrace{m[i\text{-}1, j\text{-}w_i] + v_i}_{\text{装入}})$

0-1背包 (续)

2. 递归地定义子问题最优解

- $m[i, j]$：背包容量为j，可选物品编号为1, 2, ..., i时的0-1背包问题的最优解

$$m[i,j]=\begin{cases}0, & i=0 \text{ or } j=0,\\ m[i-1,j], & 0<j<w_i,\\ \max(m[i-1,j],m[i-1,j-w_i]+v_i), & w_i\le j\le W.\end{cases}$$

0-1背包 (续)

■例：

	物品1	物品2	物品3	物品4	物品5
重量w_i	4	5	4	3	10
价值v_i	9	10	9	2	24
背包容量W	13				

➢$m[5, 13]$：考虑是否选择物品5

- 选择：$m[4, 13\text{-}10]+24 = m[4, 3]+24$
- 不选择：$m[4, 13]$

0-1背包 (续)

	物品1	物品2	物品3	物品4	物品5
重量w_i	4	5	4	3	10
价值v_i	9	10	9	2	24
背包容量W	13				

$m[i, j]$	j=0	1	2	3	4	5	6	7	8	9	10	11	12	13
i = 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	9	9	9	9	9	9	9	9	9	9
2	0	0	0	0	9	10	10	10	10	19	19	19	19	19
3	0	0	0	0	9	10	10	10	18	19	19	19	19	28
4	0	0	0	2	9	10	10	11	18	19	19	20	21	28
5	0	0	0	2	9	10	10	11	18	19	24	24	24	28

$$m[i, j] = \max(m[i\text{-}1, j],\quad m[i\text{-}1, j\text{-}w_i] + v_i)$$

0-1背包 (续)

3. 自底向上计算最优解的值

- $s[i, j]$：记录背包容量为j时编号为i的物品是否选择
  - 1——选择
  - 0——不选择

$\Theta(nW)$

```
KNAPSACK(w, v, W)
n ← v.length
let m[0..n, 0..W] and s[1..n, 1..W] be new tables
for i ← 1 to n do
    m[i, 0] ← 0
for j ← 1 to W do
    m[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j ≥ w[i] and m[i-1, j] < m[i-1, j-w[i]] + v[i]
            m[i, j] ← m[i-1, j-w[i]] + v[i]
            s[i, j] ← 1
        else m[i, j] ← m[i-1, j]
            s[i, j] ← 0
return m and s
```

0-1背包 (续)

4. 构造最优解

```
CONSTRUCT_KNAPSACK(w, W, s)
n ← w.length;  K ← W
for i ← n downto 1 do
    if K ≤ 0
        return
    if s[i, K] = 1
        print “选择编号”i“物品”
        K ← K – w[i]
```

本章小结

- 动态规划求解步骤：定义子问题、递归定义子问题最优解、自底向上求解、构建原问题最优解
- 动态规划要素：最优子结构、重叠子问题
- 求解方法：自底向上求解、带备忘的自顶向下方法求解
- 具体应用：钢条切割、矩阵链乘法最优括号化、多边形最佳三角剖分、最长公共子序列、最优二叉搜索树、0-1背包

第4章 贪心算法

苏州大学 计算机科学与技术学院
汪笑宇
Email: xywang21@suda.edu.cn

引入

贪婪，我找不到一个更好的词来描述它，它就是好！它就是对！它就是有效！

——影片《华尔街》
美国演员迈克尔.道格拉斯

本章内容

- 贪心算法（教材Chapter 16）
  - 活动选择问题
  - 贪心算法原理
  - 分数背包问题
  - Huffman编码
  - 拟阵
  - 其他应用

贪心算法求解实例

■最优解：

- 调度问题
  - 活动选择问题（教材Chapter 16.1）
  - 任务调度问题（教材Chapter 16.5）
- 图算法
  - 最小生成树（教材Chapter 23）
  - 单源点最短路径Dijkstra算法（教材Chapter 24.3）
- 其他：Huffman编码（教材Chapter 16.3）

■近似解：

- 旅行商问题TSP（教材Chapter 35.2）
- 集合覆盖问题（教材Chapter 35.3）
- 子集和问题（教材Chapter 35.5）

启发式算法 (Heuristic algorithms)

- In mathematical programming, a heuristic algorithm is a procedure that determines near-optimal solutions to an optimization problem. However, this is achieved by trading optimality, completeness, accuracy, or precision for speed. Nevertheless, heuristics is a widely used technique for a variety of reasons:
  - Problems that do not have an exact solution or for which the formulation is unknown
  - The computation of a problem is computationally intensive
  - Calculation of bounds on the optimal solution in branch and bound solution processes

来源：https://optimization.cbe.cornell.edu/index.php?title=Heuristic_algorithms

贪心算法概述

- 求最优解的问题可看作是通过一系列步骤，每一步有一个选择的集合，对于较简单的问题，动态规划显得过于复杂，可用较简单有效的算法求解
- 贪心算法总是在当前步骤上选取最好的方案，即它是一种局部最优的选择，并希望它导致一个全局最优，但有时（或者是大部分）不可能导致全局最优
  - 例：求v_i到v_j的一条最短路径，若贪心地从v_i到最近距离点v_k，未必包含在v_i到v_j的最短路径中
- 但仍有许多问题贪心法将产生全局最优解，如最小生成树MST、单源最短路径等

贪心算法概述 (续)

■一般来说，贪心算法可解的问题有如下特性：

1. 优化问题，有一个候选对象集合，如零钱、边(Kruskal)、路径(Dijkstra)、顶点(Prim)等
2. 随着算法的进行，累积形成两个集合，一个是已经被选中的对象集合，另一个是被抛弃的对象集合
3. 函数1（solution function）：检查候选对象集合是否提供了问题的解，不考虑此时的解决方法是否最优
4. 函数2：检查候选对象是否可加入到当前解的对象集合中（可行的，feasible），不考虑解决方法的最优性
5. 选择函数：指出哪个剩余的候选对象（没有被选择过也没有被丢弃过）最有可能构成问题的解
6. 目标函数：给出解的值。如零钱个数，路径长度，顶点个数等

贪心算法概述 (续)

■贪心算法一般形式

GREEDY(C)   // C是候选对象集合
$S \leftarrow \emptyset$ // 在集合S中构造解
while $C \neq \emptyset$ and not solution(S) **do**
    $x \leftarrow$ select(C)
    $C \leftarrow C \setminus \{x\}$
    if feasible($S \cup \{x\}$)
        $S \leftarrow S \cup \{x\}$
if solution(S)
    return S
else return “No solution”

贪心算法概述 (续)

- 例：找零问题：设数组$A[1..n]$中的元素表示n个零钱面值，需寻找可找开某个金额M的最少零钱数量，及相应找零方案

```
CHANGE_GREEDY(A, M)
  S ← ∅;  s ← 0    // S为解中包含的零钱集合，s为S中零钱面值之和
  while s ≠ M do
      x ← 使得s+x不超过M的最大零钱面值
      if 找不到这样的面值
          error “无法找开”
      else S ← S ∪ {一个x面值的零钱}
          s ← s + x
   return S
```

贪心算法得到的是否是最优解？
在正常的货币系统下是最优解，但零钱面值任意指定时不一定最优

贪心算法概述 (续)

■找零问题特性：

1. 找最少零钱数，候选对象集为面值在 $A[1..n]$中的零钱
2. 选择的零钱集合和未被选的零钱集合
3. 判断解函数，检查目前已选零钱集合中的金额是否等于要找的钱数
4. 如果集合中零钱面额不超过应找金额，则该集合是可行的
5. 选择函数，从未选零钱集合中找面值最大的零钱
6. 目标函数，计算零钱数目

贪心算法内容

- ■活动选择问题
- ■贪心算法原理
- ■分数背包问题
- ■Huffman编码
- ■拟阵
- ■其他应用

活动选择问题

- 多个活动竞争资源的调度问题：尽可能多地选择互不冲突的活动
- 设有n个活动（activity）$S=\{a_1, a_2, \ldots, a_n\}$，均要使用某资源（如教室），该资源使用方式为独占式，一次只供一个活动使用
  - 每个活动a_i发生的时间为$[s_i, f_i)$, $0 \le s_i < f_i < \infty$
  - 两活动a_i, a_j兼容（compatible不冲突）：$[s_i, f_i)$, $[s_j, f_j)$不重叠，满足$s_i \ge f_j$ 或$s_j \ge f_i$，即：一活动的开始时间大于等于另一活动的完成时间
  - 活动选择问题：选择最多的互不冲突的活动，使兼容活动集合最大，即求解$A \subseteq S$，A中活动互不冲突且$|A|$最大

活动选择问题 (续)

■假定活动已按结束时间单调递增顺序排序

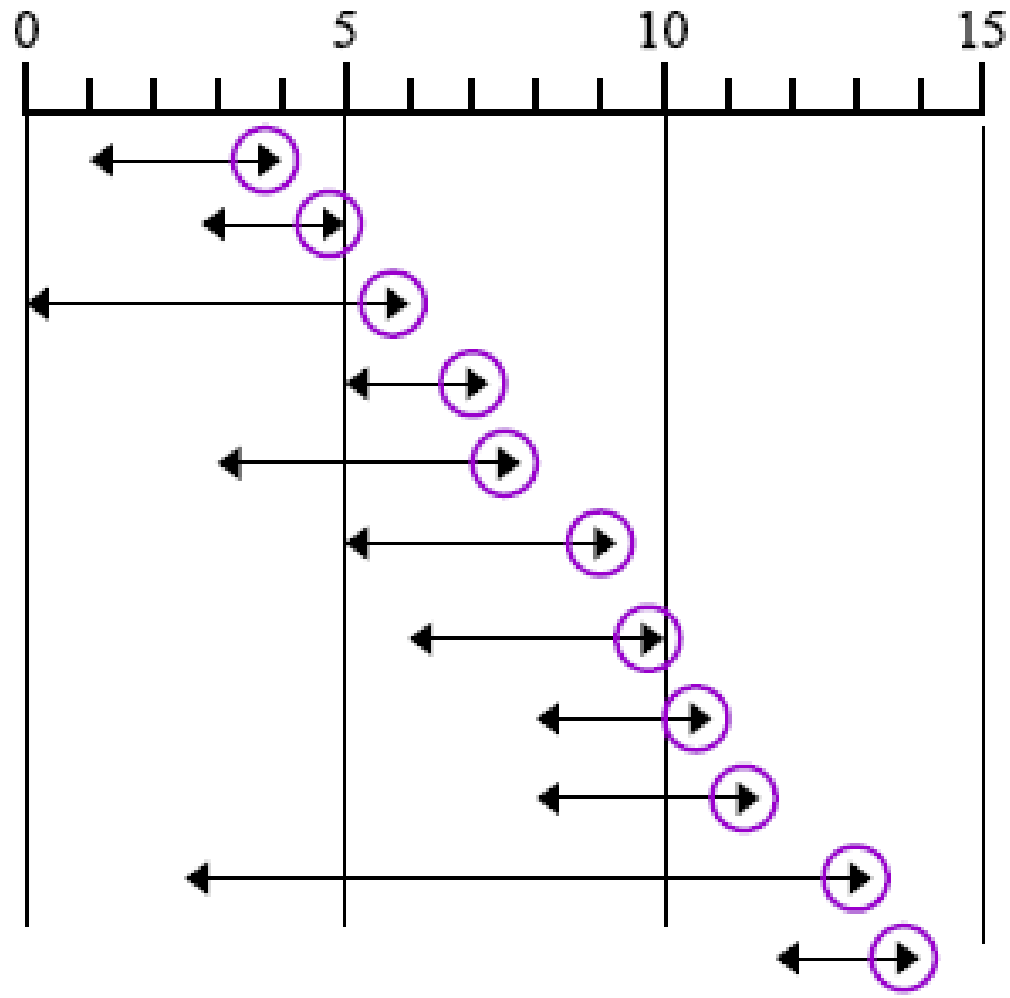
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- 问题的解：$A_1=\{a_3, a_9, a_{11}\}$，$A_2=\{a_1, a_4, a_8, a_{11}\}$，$A_3=\{a_2, a_4, a_9, a_{11}\}$
- 最优解：A_2和A_3

■此问题可用迭代方法直接给出贪心算法，但为比较和动态规划关系，以下先考虑动态规划解法

活动选择问题 (续)

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



活动选择问题 (续)

1. 活动选择问题的最优子结构

- S_{ij}：在a_i结束之后开始且在a_j开始之前结束的活动集合
  $S_{ij} = \{a_k \in S: f_i \le s_k < f_k \le s_j\}$
- A_{ij}：S_{ij}的一个最大相互兼容的活动子集，包含活动a_k（A_{ij}是S_{ij}问题的最优解）
- 问题分解：
  S_{ik}中最大兼容活动子集 + S_{kj}中最大兼容活动子集
  $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ （剪切-粘贴及反证法证明）

活动选择问题 (续)

2. 递归解

- $c[i, j]$：在a_i结束之后开始且在a_j开始之前结束的活动集合S_{ij}的一个最大相互兼容的活动子集的大小，即$c[i, j] = |A_{ij}|$
- 若$a_k \in A_{ij}$，则有$c[i, j] = c[i, k] + c[k, j] + 1$
- 总体递归式：

$$c[i,j] = \begin{cases} 0, & S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}, & S_{ij} \neq \emptyset. \end{cases}$$

活动选择问题 (续)

3. 贪心算法

- 直观上，我们应该选择这样一个活动，选出它后剩下的资源应能被尽量多的其他任务所用
- 每次选择候选集中最早结束的活动
- **定理16.1** 考虑任意非空子问题S_{ij}，令a_m是S_{ij}中结束时间最早的活动，即：$f_m = \min\{f_k: a_k \in S_{ij}\}$，则
  1. a_m在S_{ij}的某个最大兼容活动子集中
  2. 子问题S_{im}的解是空集

活动选择问题 (续)

3. 贪心算法

➢**定理16.1** 考虑任意非空子问题S_{ij}，令a_m是S_{ij}中结束时间最早的活动，即：$f_m = \min\{f_k: a_k \in S_{ij}\}$，则

1. a_m在S_{ij}的某个最大兼容活动子集中
2. 子问题S_{im}的解是空集

➢证明：（第2部分，反证法）假定S_{im}的解非空，则存在$a_k \in S_{im}$，使得$f_i \le s_k < f_k \le s_m$。由此得到$a_k \in S_{ij}$的完成时间先于$a_m$，与$a_m$是$S_{ij}$最早完成的活动矛盾

活动选择问题 (续)

3. 贪心算法

- **定理16.1** 考虑任意非空子问题S_{ij}，令a_m是S_{ij}中结束时间最早的活动，即：$f_m = \min\{f_k: a_k \in S_{ij}\}$，则
  1. a_m在S_{ij}的某个最大兼容活动子集中
  2. 子问题S_{im}的解是空集
- 证明：（第1部分）设A_{ij}是S_{ij}的某个最优解，假设A_{ij}中的活动已按完成时间单调递增排序，且a_k是A_{ij}中最早结束的活动：
  1、若a_k=a_m，则问题已得证，即最优解包含a_m；
  2、若$a_k \neq a_m$，构造子集$A'_{ij} = (A_{ij} - \{a_k\}) \cup \{a_m\}$，即将最优解中的$a_k$替换为$a_m$，则需证明$A'_{ij}$也是最优解
  因为$f_m \leq f_k$，因此A'_{ij}中的活动也不冲突，且$|A'_{ij}| = |A_{ij}|$
  A'_{ij}也是S_{ij}的一个最优解，包含a_m

活动选择问题 (续)

3. 贪心算法

- 动态规划求解时，原问题S_{ij}可分解为两个子问题S_{ik}和S_{kj}求解，且这种分解有$|S_{ij}|$种可能
- 定理16.1可简化问题求解过程：
  - 求S_{ij}最优解时只用到一个子问题，另一个子问题为空
  - 只需考虑一种选择，即选择S_{ij}中最早完成的活动
- 定理16.1可以自顶向下的方式解每一个子问题

活动选择问题 (续)

3. 贪心算法

- 当某个a_m加入解集合后，我们总是在剩余活动中选择第一个不与a_m冲突的活动加入解集，该活动是能够最早完成且与a_m兼容的
- 这种选择为剩余活动的调度留下了尽可能多的机会，即：留出尽可能多的时间给剩余的尚未调度的活动，以使解集合中包含的活动最多

每次选一个最早完成并与刚加入解集元素兼容的活动

活动选择问题 (续)

3. 贪心算法

➢例：

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

1. 选择a_1，则下一活动开始时间须大于等于4，排除a_2, a_3
2. 选择a_4，则下一活动开始时间须大于等于7，排除a_5, a_6, a_7
3. 选择a_8，则下一活动开始时间须大于等于11，排除a_9, a_{10}
4. 选择a_{11}，则下一活动开始时间须大于等于14
5. 无可选活动，结束，解集为$\{a_1, a_4, a_8, a_{11}\}$

活动选择问题 (续)

4. 递归的贪心算法

- 输入：$s[1..n]$：活动开始时间
  $f[0..n]$：活动结束时间
  i, j：子问题S_{ij}的下标
- 输出：S_{ij}的最优解

注：定义虚拟活动a_0结束时间为f_0
活动已按照结束时间单调递增排序

```
RECURSIVE_ACTIVITY_SELECTOR(s, f, i, j)
m ← i + 1
while m ≤ j and s[m] < f[i] do   // 将与a_i冲突的活动去掉
    m ← m + 1
if m ≤ j
    return {a_m} ∪ RECURSIVE_ACTIVITY_SELECTOR(s, f, m, j)
else return ∅
```

RECURSIVE_ACTIVITY_SELECTOR(s, f, 0, n)的运行时间为$\Theta(n)$

活动选择问题 (续)

5. 迭代贪心算法

- RECURSIVE_ACTIVITY_SELECTOR几乎就是尾递归：以一个对自身的递归调用再接一次并集操作结尾
- 尾递归过程改为迭代形式通常很直接，某些特定语言的编译器可以自动完成这一工作

```
GREEDY_ACTIVITY_SELECTOR(s, f)
    n ← s.length
    A ← {a_1}
    k ← 1
    for m ← 2 to n do
        if s[m] ≥ f[k]
            A ← A ∪ {a_m}
            k ← m
    return A
```

时间复杂度：$\Theta(n)$（已排序情况）

排序：$\Theta(n\lg n)$

算法正确性证明？
循环不变式及证明→定理16.1证明→算法正确